Chapter 2. Introducing SOAP

SOAP's place in the web services technology stack is as a standardized packaging protocol for the messages shared by applications. The specification defines nothing more than a simple XML-based envelope for the information being transferred, and a set of rules for translating application and platform-specific data types into XML representations. SOAP's design makes it suitable for a wide variety of application messaging and integration patterns. This, for the most part, contributes to its growing popularity.

This chapter explains the parts of the SOAP standard. It covers the message format, the exception-reporting mechanism (*faults*), and the system for encoding values in XML. It discusses using SOAP over transports that aren't HTTP, and concludes with thoughts on the future of SOAP. You'll learn what SOAP does and how it does it, and get a firm understanding of the flexibility of SOAP. Later chapters build on this to show how to program with SOAP using toolkits that abstract details of the XML.

2.1 SOAP and XML

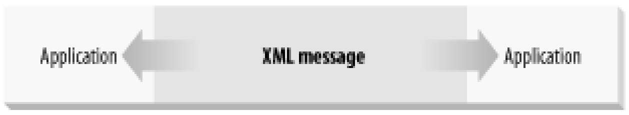
SOAP is XML. That is, SOAP is an application of the XML specification. It relies heavily on XML standards like XML Schema and XML Namespaces for its definition and function. If you are not familiar with any of these, you'll probably want to get up to speed before continuing with the information in this chapter (you can find information about each of these specifications at the World Wide Web Consortium's web site at http://www.w3c.org/). This book assumes you are familiar with these specifications, at least on a cursory level, and will not spend time discussing them. The only exception is a quick introduction to the XML Schema data types in Appendix B.

2.1.1 XML Messaging

XML messaging is where applications exchange information using XML documents (see Figure 2-1). It provides a flexible way for applications to communicate, and forms the basis of SOAP.

A message can be anything: a purchase order, a request for a current stock price, a query for a search engine, a listing of available flights to Los Angeles, or any number of other pieces of information that may be relevant to a particular application.

Figure 2-1. XML messaging



Because XML is not tied to a particular application, operating system, or programming language, XML messages can be used in all environments. A Windows Perl program can create an XML document representing a message, send it to a Unix-based Java program, and affect the behavior of that Java program.

The fundamental idea is that two applications, regardless of operating system, programming language, or any other technical implementation detail, may openly share information using nothing more than a simple message encoded in a way that both applications understand. SOAP provides a standard way to structure XML messages.

2.1.2 RPC and EDI

XML messaging, and therefore SOAP, has two related applications: RPC and EDI. Remote Procedure Call (RPC) is the basis of distributed computing, the way for one program to make a procedure (or function, or method, call it what you will) call on another, passing arguments and receiving return values. Electronic Document Interchange (EDI) is basis of automated business transactions, defining a standard format and interpretation of financial and commercial documents and messages.

If you use SOAP for EDI (known as "document-style" SOAP), then the XML will be a purchase order, tax refund, or similar document. If you use SOAP for RPC (known, unsurprisingly, as "RPC-style" SOAP) then the XML will be a representation of parameter or return values.

2.1.3 The Need for a Standard Encoding

If you're exchanging data between heterogeneous systems, you need to agree on a common representation. As you can see in Example 2-1, a single piece of data like a telephone number may be represented in many different, and equally valid ways in XML.

Example 2-1. Many XML representations of a phone number

```
<phoneNumber>(123) 456-7890</phoneNumber>
<phoneNumber>
    <areaCode>123</areaCode>
    <exchange>456</exchange>
    <number>7890</number>
</phoneNumber>
<phoneNumber area="123"  exchange="456"  number="7890" />
<phone area="123">
    <exchange>456</exchange>
    <number>7890</number>
</phone>
```

Which is the correct encoding? Who knows! The correct one is whatever the application is expecting. In other words, simply saying that server and client are using XML to exchange information is not enough. We need to define:

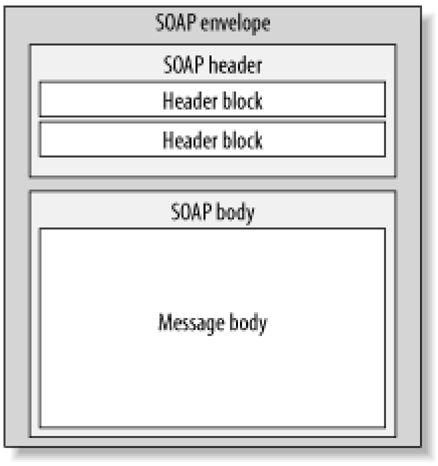
- The types of information we are exchanging
- How that information is to be expressed as XML
- How to actually go about sending that information

Without these agreed conventions, programs cannot know how to decode the information they're given, even if it's encoded in XML. SOAP provides these conventions.

2.2 SOAP Messages

A SOAP message consists of an envelope containing an optional header and a required body, as shown in Figure 2-2. The header contains blocks of information relevant to how the message is to be processed. This includes routing and delivery settings, authentication or authorization assertions, and transaction contexts. The body contains the actual message to be delivered and processed. Anything that can be expressed in XML syntax can go in the body of a message.

Figure 2-2. The SOAP message structure



The XML syntax for expressing a SOAP message is based on the `http://www.w3.org/2001/06/soap-envelope` namespace. This XML namespace identifier points to an XML Schema that defines the structure of what a SOAP message looks like.

If you were using document-style SOAP, you might transfer a purchase order with the XML in Example 2-2.

Example 2-2. A purchase order in document-style SOAP

```
<s:Envelope
 xmlns:s="http://www.w3.org/2001/06/soap-envelope">
    <s:Header>
       <m:transaction xmlns:m="soap-transaction"
                      s:mustUnderstand="true">
           <transactionID>1234</transactionID>
       </m:transaction>
    </s:Header>
    <s:Body>
       <n:purchaseOrder xmlns:n="urn:OrderService">
          <from><person>Christopher Robin</person>
                <dept>Accounting</dept></from>
          <to><person>Pooh Bear</person>
               <dept>Honey</dept></to>
          <order><quantity>1</quantity>
                 <item>Pooh Stick</item></order>
       </n:purchaseOrder>
    </s:Body>
</s:Envelope>
```

This example illustrates all of the core components of the SOAP Envelope specification. There is the `<s:Envelope>`, the topmost container that comprises the SOAP message; the optional `<s:Header>`, which contains additional blocks of information about how the body payload is to be processed; and the mandatory `<s:Body>` element that contains the actual message to be processed.

2.2.1 Envelopes

Every `Envelope` element must contain exactly one `Body` element. The `Body` element may contain as many child nodes as are required. The contents of the `Body` element are the message. The `Body` element is defined in such a way that it can contain any valid, well-formed XML that has been namespace qualified and does not contain any processing instructions or Document Type Definition (DTD) references.

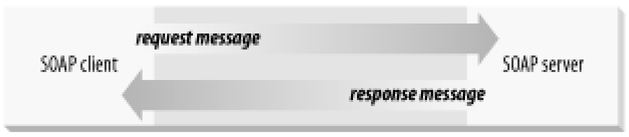
If an `Envelope` contains a `Header` element, it must contain no more than one, and it must appear as the first child of the `Envelope`, beforethe `Body`. The header, like the body, may contain any valid, well-formed, and namespace-qualified XML that the creator of the SOAP message wishes to insert.

Each element contained by the `Header` is called a *header block*. The purpose of a header block is to communicate contextual information relevant to the processing of a SOAP message. An example might be a header block that contains authentication credentials, or message routing information. Header blocks will be highlighted and explained in greater detail throughout the remainder of the book. In Example 2-2, the header block indicates that the document has a transaction ID of "1234".

2.2.2 RPC Messages

Now let's see an RPC-style message. Typically messages come in pairs, as shown in Figure 2-3: the request (the client sends function call information to the server) and the response (the server sends return value(s) back to the client). SOAP doesn't require every request to have a response, or vice versa, but it is common to see the request-response pairing.

Figure 2-3. Basic RPC messaging architecture



Imagine the server offers this function, which returns a stock's price, as a SOAP service:

```
public Float getQuote(String symbol);
```

Example 2-3 illustrates a simple RPC-style SOAP message that represents a request for IBM's current stock price. Again, we show a header block that indicates a transaction ID of "1234".

Example 2-3. RPC-style SOAP message

```
<s:Envelope
 xmlns:s="http://www.w3.org/2001/06/soap-envelope">
    <s:Header>
       <m:transaction xmlns:m="soap-transaction"
                      s:mustUnderstand="true">
           <transactionID>1234</transactionID>
       </m:transaction>
    </s:Header>
    <s:Body>
       <n:getQuote xmlns:n="urn:QuoteService">
          <symbol xsi:type="xsd:string">
             IBM
          </symbol>
       </n:getQuote>
    </s:Body>
</s:Envelope>
```

Example 2-4 is a possible response that indicates the operation being responded to and the requested stock quote value.

Example 2-4. SOAP response to request in Example 2-3

```
<s:Envelope
 xmlns:s="http://www.w3.org/2001/06/soap-envelope">
    <s:Body>
       <n:getQuoteRespone
            xmlns:n="urn:QuoteService">
          <value xsi:type="xsd:float">
             98.06
          </value>
       </n:getQuoteResponse>
    </s:Body>
</s:Envelope>
```

2.2.3 The mustUnderstand Attribute

When a SOAP message is sent from one application to another, there is an implicit requirement that the recipient must understand how to process that message. If the recipient does not understand the message, the recipient must reject the message and explain the problem to the sender. This makes sense: if Amazon.com sent O'Reilly a purchase order for 150 electric drills, someone from O'Reilly would call someone from Amazon.com and explain that O'Reilly and Associates sells books, not electric drills.

Header blocks are different. A recipient may or may not understand how to deal with a particular header block but still be able to process the primary message properly. If the sender of the message wants to require that the recipient understand a particular block, it may add a `mustUnderstand="true"` attribute to the header block. If this flag is present, and the recipient does not understand the block to which it is attached, the recipient must reject the entire message.

In the `getQuote` envelope we saw earlier, the `transaction` header contains the `mustUnderstand="true"` flag. Because this flag is set, regardless of whether or not the recipient understands and is capable of processing the message body (the `getQuote` message), if it does not understand how to deal with the `transaction` header block, the entire message must be rejected. This guarantees that the recipient understands transactions.

2.2.4 Encoding Styles

As part of the overall specification, Section 5 of the SOAP standard introduces a concept known as *encoding styles*. An encoding style is a set of rules that define exactly how native application and platform data types are to be encoded into a common XML syntax. These are, obviously, for use with RPC-style SOAP.

The encoding style for a particular set of XML elements is defined through the use of the `encodingStyle` attribute, which can be placed anywhere in the document and applies to all subordinate children of the element on which it is located.

For example, the `encodingStyle` attribute on the `getQuote` element in the body of Example 2-5 indicates that all children of the `getQuote` element conform to the encoding style rules defined in Section 5.

Example 2-5. The encodingStyle attribute

```
<s:Envelope
 xmlns:s="http://www.w3.org/2001/06/soap-envelope">
 <s:Body>
  <n:getQuote xmlns:n="urn:QuoteService"
   s:encodingStyle="http://www.w3.org/2001/06/soap-encoding">
    <symbol xsi:type="xsd:string">IBM</symbol>
  </n:getQuote>
 </s:Body>
</s:Envelope>
```

Even though the SOAP specification defines an encoding style in Section 5, it has been explicitly declared that no single style is the default serialization scheme. Why is this important?

Encoding styles are how applications on different platforms share information, even though they may not have common data types or representations. The approach that the SOAP Section 5 encoding style takes is just one possible mechanism for providing this, but it is not suitable in every situation.

For example, in the case where a SOAP message is used to exchange a purchase order that already has a defined XML syntax, there is no need for the Section 5 encoding rules to be applied. The purchase order would simply be dropped into the `Body` section of the SOAP envelope as is.

The SOAP Section 5 encoding style will be discussed in much greater detail later in this chapter, as most SOAP applications and libraries use it.

2.2.5 Versioning

There have been several versions of the SOAP specification put into production. The most recent working draft, SOAP Version 1.2, represents the first fruits of the World Wide Web

Consortium's (W3C) effort to standardize an XML-based packaging protocol for web services. The W3C chose SOAP as the basis for that effort.

The previous version of SOAP, Version 1.1, is still widely used. In fact, at the time we are writing this, there are only three implementations of the SOAP 1.2 specification available: SOAP::Lite for Perl, Apache SOAP Version 2.2, and Apache Axis (which is not even in beta status).

While SOAP 1.1 and 1.2 are largely the same, the differences that do exist are significant enough to warrant mention. To prevent subtle incompatibility problems, SOAP 1.2 introduces a versioning model that deals with how SOAP Version 1.1 processors and SOAP Version 1.2 processors may interact. The rules for this are fairly straightforward:

1. If a SOAP Version 1.1 compliant application receives a SOAP Version 1.2 message, a "version mismatch" error will be triggered.
2. If a SOAP Version 1.2 compliant application receives a SOAP Version 1.1 message, the application may choose to either process it according to the SOAP Version 1.1 specification or trigger a "version mismatch" error.

The version of a SOAP message can be determined by checking the namespace defined for the SOAP envelope. Version 1.1 uses the namespace `http://schemas.xmlsoap.org/soap/envelope/`, whereas Version 1.2 uses the namespace `http://www.w3.org/2001/06/soap-envelope`. Example 2-6 illustrates the difference.

Example 2-6. Distinguishing between SOAP 1.1 and SOAP 1.2

```
<!-- Version 1.1 SOAP Envelope -->
<s:Envelope
 xmlns:s="
http://schemas.xmlsoap.org/soap/envelope/">
  ...
</s:Envelope>

<!-- Version 1.2 SOAP Envelope -->
<s:Envelope
 xmlns:s="
http://www.w3.org/2001/06/soap-envelope">
  ...
</s:Envelope>
```

When applications report a version mismatch error back to the sender of the message, it may optionally include an `Upgrade` header block that tells the sender which version of SOAP it supports. Example 2-7 shows the `Upgrade` header in action.

Example 2-7. The Upgrade header

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Header>
    <V:Upgrade xmlns:V="http://www.w3.org/2001/06/soap-upgrade">
      <envelope qname="ns1:Envelope"
                xmlns:ns1="http://www.w3.org/2001/06/soap-envelope"/>
    </V:Upgrade>
  </s:Header>
  <s:Body>
    <s:Fault>
      <faultcode>s:VersionMismatch</faultcode>
      <faultstring>Version Mismatch</faultstring>
    </s:Fault>
  </s:Body>
</s:Envelope>
```

For backwards compatibility, version mismatch errors must conform to the SOAP Version 1.1 specification, regardless of the version of SOAP being used.

2.3 SOAP Faults

A SOAP fault (shown in Example 2-8) is a special type of message specifically targeted at communicating information about errors that may have occurred during the processing of a SOAP message.

Example 2-8. SOAP fault

```
<s:Envelope xmlns:s="...">
   <s:Body>
      <s:Fault>
       <faultcode>Client.Authentication</faultcode>
       <faultstring>
          Invalid credentials
       </faultstring>
       <faultactor>http://acme.com</faultactor>
       <details>
          <!-- application specific details -->
       </details>
      </s:Fault>
   </s:Body>
</s:Envelope>
```

The information communicated in the SOAP fault is as follows:

The fault code

An algorithmically generated value for identifying the type of error that occurred. The value must be an XML Qualified Name, meaning that the name of the code only has meaning within a defined XML namespace.

The fault string

A human-readable explanation of the error.

The fault actor

The unique identifier of the message processing node at which the error occurred (actors will be discussed later).

The fault details

Used to express application-specific details about the error that occurred. This must be present if the error that occurred is directly related to some problem with the body of the message. It must not be used, however, to express information about errors that occur in relation to any other aspect of the message process.

2.3.1 Standard SOAP Fault Codes

SOAP defines four standard types of faults that belong to the `http://www.w3.org/2001/06/soap-envelope` namespace. These are described here:

`VersionMismatch`

The SOAP envelope is using an invalid namespace for the SOAP Envelope element.

`MustUnderstand`

A `Header` block contained a `mustUnderstand="true"` flag that was not understood by the message recipient.

`Server`

An error occurred that can't be directly linked to the processing of the message.

`Client`

There is a problem in the message. For example, the message contains invalid authentication credentials, or there is an improper application of the Section 5 encoding style rules.

These fault codes can be extended to allow for more expressive and granular types of faults, while still maintaining backwards compatibility with the core fault codes.

The example SOAP fault demonstrates how this extensibility works. The `Client.Authentication` fault code is a more granular derivative of the `Client` fault type. The "." notation indicates that the piece to the left of the period is more generic than the piece that is to the right of the period.

2.3.2 MustUnderstand Faults

As mentioned earlier, a header block contained within a SOAP message may indicate through the `mustUnderstand="true"` flag that the recipient of the message must understand how to process the contents of the header block. If it cannot, then the recipient must return a `MustUnderstand` fault back to the sender of the message. In doing so, the fault should communicate specific information about the header blocks that were not understood by the recipient.

The SOAP fault structure is not allowed to express any information about which headers were not understood. The `details` element would be the only place to put this information and it is reserved solely for the purpose of expressing error information related to the processing of the body, not the header.

To solve this problem, the SOAP Version 1.2 specification defines a standard `Misunderstood` header block that can be added to the SOAP fault message to indicate which header blocks in the received message were not understood. Example 2-9 shows this.

Example 2-9. The Misunderstood header

```
<s:Envelope xmlns:s="...">
   <s:Header>
      <f:Misunderstood qname="abc:transaction"
                       xmlns:="soap-transactions" />
   </s:Header>
   <s:Body>
      <s:Fault>
       <faultcode>MustUnderstand</faultcode>
       <faultstring>
            Header(s) not understood
       </faultstring>
       <faultactor>http://acme.com</faultactor>
      </s:Fault>
   </s:Body>
</s:Envelope>
```

The `Misunderstood` header block is optional, which makes it unreliable to use as the primary method of determining which headers caused the message to be rejected.

2.3.3 Custom Faults

A web service may define its own custom fault codes that do not derive from the ones defined by SOAP. The only requirement is that these custom faults be namespace qualified. Example 2-10 shows a custom fault code.

Example 2-10. A custom fault

```
<s:Envelope xmlns:s="...">
   <s:Body>
      <s:Fault xmlns:xyz="urn:myCustomFaults">
         <faultcode>xyz:CustomFault</faultcode>
         <faultstring>
             My custom fault!
         </faultstring>
      </s:Fault>
   </s:Body>
</s:Envelope>
```

Approach custom faults with caution: a SOAP processor that only understands the standard four fault codes will not be able to take intelligent action upon receipt of a custom fault.

However, custom faults can still be useful in situations where the standard fault codes are too generic or are otherwise inadequate for the expression of what error occurred.

For the most part, the extensibility of the existing four fault codes makes custom fault codes largely unnecessary.

2.4 The SOAP Message Exchange Model

Processing a SOAP message involves pulling apart the envelope and doing something with the information that it carries. SOAP defines a general framework for such processing, but leaves the actual details of how that processing is implemented up to the application.

What the SOAP specification does have to say about message processing deals primarily with how applications exchange SOAP messages. Section 2 of the specification outlines a very specific message exchange model.

2.4.1 Message Paths and Actors

At the core of this exchange model is the idea that while a SOAP message is fundamentally a one-way transmission of an envelope from a sender to a receiver, that message may pass through various intermediate processors that each in turn do something with the message. This is analogous to a Unix pipeline, where the output of one program becomes the input to another, and so on until you get the output you want.

A SOAP intermediary is a web service specially designed to sit between a service consumer and a service provider and add value or functionality to the transaction between the two. The set of intermediaries that the message travels through is called the message path. Every intermediary along that path is known as an actor.

The construction of a message path (the definition of which nodes a message passes through) is not covered by the SOAP specification. Various extensions to SOAP, such as Microsoft's SOAP Routing Protocol (WS-Routing) have emerged to fill that gap, but there is still no standard (de facto or otherwise) method of expressing the message path. We cover WS-Routing later.

What SOAP does specify, however, is a mechanism of identifying which parts of the SOAP message are intended for processing by specific actors in its message path. This mechanism is known as "targeting" and can only be used in relation to header blocks (the body of the SOAP envelope cannot be explicitly targeted at a particular node).

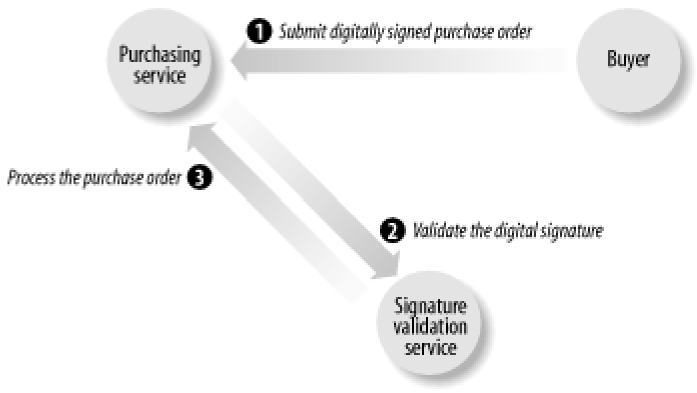
A header block is targeted to a specific actor on its message path through the use of the special `actor` attribute. The value of the `actor` attribute is the unique identifier of the intermediary being targeted. This identifier may be the URL where the intermediary may be found, or something more generic. Intermediaries that do not match the `actor` attribute must ignore the header block.

For example, imagine that I am a wholesaler of fine cardigan sweaters. I set up a web service that allows me to receive purchase orders from my customers in the form of SOAP messages. You, one of my best customers, want to submit an order for 100 sweaters. So you send me a SOAP message that contains the purchase order.

For our mutual protection, however, I have established a relationship with a trusted third-party web service that can help me validate that the purchase order you sent really did come from you. This service works by verifying that your digital signature header block embedded in the SOAP message is valid.

When you send that message to me, it is going to be routed through this third-party signature verification service, which will, in turn, extract the digital signature, validate it, and add a new header block that tells me whether the signature is valid. The transaction is depicted in Figure 2-4.

Figure 2-4. The signature validation intermediary


Purchasing service
1 Submit digitally signed purchase order
Buyer
Process the purchase order 3
2 Validate the digital signature
Signature validation service

Now, the signature verification intermediary needs to have some way of knowing which header block contains the digital signature that it is expected to verify. This is accomplished by targeting the digital signature block to the verification service, as in Example 2-11.

Example 2-11. The actor header

```
<s:Envelope xmlns:s="...">
   <s:Header>
      <x:signature actor="uri:SignatureVerifier">
         ...
      </x:signature>
   </s:Header>
   <s:Body>
      <abc:purchaseOrder>...</abc:purchaseOrder>
   </s:Body>
</s:Envelope>
```

The `actor` attribute on the `signature` header block is how the signature verifier intermediary knows that it is responsible for processing that header block. If the message does not pass through the signature verifier, then the signature block is ignored.

2.4.2 The SOAP Routing Protocol

Remember, SOAP does not specify howthe message is to be routed to the signature verification service, only that it should be at some point during the processing of the SOAP message. This makes the implementation of SOAP message paths a fairly difficult proposition since there is no single standard way of representing that path. The SOAP Routing Protocol (WS-Routing) is Microsoft's proposal for solving this problem.

WS-Routing defines a standard SOAP header block (see Example 2-12) for expressing routing information. Its role is to define the exact sequence of intermediaries through which a message is to pass.

Example 2-12. A WS-Routing message

```
<s:Envelope xmlns:s="...">
 <s:Header>
  <m:path xmlns:m="http://schemas.xmlsoap.org/rp/"
          s:mustUnderstand="true">
   <m:action>http://www.im.org/chat</m:action>
    <m:to>http://D.com/some/endpoint</m:to>
    <m:fwd>
     <m:via>http://B.com</m:via>
     <m:via>http://C.com</m:via>
    </m:fwd>
    <m:rev>
     <m:via/>
    </m:rev>
    <m:from>mailto:johndoe@acme.com</m:from>
    <m:id>
      uuid:84b9f5d0-33fb-4a81-b02b-5b760641c1d6
    </m:id>
  </m:path>
 </S:Header>
 <S:Body>
  ...
 </S:Body>
</S:Envelope>
```

In this example, we see the SOAP message is intended to be delivered to a recipient located at `http://d.com/some/endpoint` but that it must first go through both the `http://b.com` and `http://c.com` intermediaries.

To ensure that the message path defined by the WS-Routing header block is properly followed, and because WS-Routing is a third-party extension to SOAP that not every SOAP processor will understand, the `mustUnderstand="true"` flag can be set on the `path` header block.

2.5 Using SOAP for RPC-Style Web Services

RPC is the most common application of SOAP at the moment. The following sections show how method calls and return values are encoded in SOAP message bodies.

2.5.1 Invoking Methods

The rules for packaging an RPC request in a SOAP envelope are simple:

- The method call is represented as a single `structure` with each in or in-out parameter modeled as a field in that structure.
- The names and physical order of the parameters must correspond to the names and physical order of the parameters in the method being invoked.

This means that a Java method with the following signature:

```
String checkStatus(String orderCode,
                   String customerID);
```

can be invoked with these arguments:

```
result = checkStatus("abc123", "Bob's Store")
```

using the following SOAP envelope:

```
<s:Envelope xmlns:s="...">
  <s:Body>
    <checkStatus xmlns="..."
                 s:encodingStyle="http://www.w3.org/2001/06/soap-encoding">
      <orderCode xsi:type="string">abc123</orderCode>
      <customerID xsi:type="string">
        Bob's Store
      </customerID>
    </checkStatus>
  </s:Body>
</s:Envelope>
```

The SOAP RPC conventions do not require the use of the SOAP Section 5 encoding style and `xsi:type` explicit data typing. They are, however, widely used and will be what we describe.

2.5.2 Returning Responses

Method responses are similar to method calls in that the structure of the response is modeled as a single `structure` with a field for each in-out or out parameter in the method signature. If the `checkStatus` method we called earlier returned the string `new`, the SOAP response might be something like Example 2-13.

Example 2-13. Response to the method call

```
<s:Envelope xmlns:s="...">
  <s:Body>
    <checkStatusResponse
     s:encodingStyle="http://www.w3.org/2001/06/soap-encoding">
      <return xsi:type="xsd:string">new</return>
    </checkStatusResponse>
  </SOAP:Body>
</SOAP:Envelope>
```

The name of the message response structure (`checkStatusResponse)` element is not important, but the convention is to name it after the method, with `Response` appended. Similarly, the name of the return element is arbitrary—the first field in the message response structure is assumed to be the return value.

2.5.3 Reporting Errors

The SOAP RPC conventions make use of the SOAP fault as the standard method of returning error responses to RPC clients. As with standard SOAP messages, the SOAP fault is used to convey the exact nature of the error that has occurred and can be extended to provide additional information through the use of the `detail` element. There's little point in customizing error messages in SOAP faults when you're doing RPC, as most SOAP RPC implementations will not know how to deal with the custom error information.

2.6 SOAP's Data Encoding

The first part of the SOAP specification outlines a standard envelope format for packaging data. The second part of the specification (specifically, Section 5) outlines one possible method of serializing the data intended for packaging. These rules outline in specific detail how basic application data types are to be mapped and encoded into XML format when embedded into a SOAP Envelope.

The SOAP specification introduces the SOAP encoding style as "a simple type system that is a generalization of the common features found in type systems in programming languages, databases, and semi-structured data." As such, these encoding rules can be applied in nearly any programming environment regardless of the minor differences that exist between those environments.

Encoding styles are completely optional, and in many situations not useful (recall the purchase order example we gave earlier in this chapter, where it made sense to ship a document and not an encoded method call/response). SOAP envelopes are designed to carry any arbitrary XML documents no matter what the body of the message looks like, or whether it conforms to any specific set of data encoding rules. The Section 5 encoding rules are offered only as a convenience to allow applications to dynamically exchange information without a priori knowledge of the types of information to be exchanged.

2.6.1 Understanding the Terminology

Before continuing, it is important to gain a firm understanding of the vocabulary used to describe the encoding process. Of particular importance are the terms *value* and *accessor*.

A value represents either a single data unit or combination of data units. This could be a person's name, the score of a football game, or the current temperature. An accessorrepresents an element that contains or allows access to a value. In the following, `firstname` is an accessor, and `Joe` is a value:

```
<firstname> Joe </firstname>
```

A compound value represents a combination of two or more accessors grouped as children of a single accessor, and is demonstrated in Example 2-14.

Example 2-14. A compound value

```
<name>
    <firstname> Joe </firstname>
    <lastname> Smith </lastname>
</name>
```

There are two types of compound values, structs (the structures we talked about earlier) and arrays. A *struct* is a compound value in which each accessor has a different name. An *array* is a compound value in which the accessors have the same name (values are identified by their positions in the array). A struct and an array are shown in Example 2-15.

Example 2-15. Structs and arrays

```
<!--A struct -->
<person>
    <firstname>Joe</firstname>
    <lastname>Smith</lastname>
</person>

<!--An array-->
<people>
    <person name='joe smith'/>
    <person name='john doe'/>
</people>
```

Through the use of the special `id` and `href` attributes, SOAP defines that accessors may either be *single-referenced* or *multireferenced*. A single-referenced accessor doesn't have an identity except as a child of its parent element. In Example 2-16, the `<address>` element is a single-referenced accessor.

Example 2-16. A single-referenced accessor

```
<people>
   <person name='joe smith'>
      <address>
         <street>111 First Street</street>
         <city>New York</city>
         <state>New York</state>
      </address>
   </person>
</people>
```

A multireferenced accessor uses `id` to give an identity to its value. Other accessors can use the `href` attribute to refer to their values. In Example 2-17, each person has the same address, because they reference the same multireferenced `address` accessor.

Example 2-17. A multireferenced accessor

```
<people>
   <person name='joe smith'>
      <address href='#address-1'
   </person>
   <person name='john doe'>
      <address href='#address-1'
   </person>
</people>
<address id='address-1'>
   <street>111 First Street</street>
   <city>New York</city>
   <state>New York</state>
</address>
```

This approach can also be used to allow an accessor to reference external information sources that are not a part of the SOAP Envelope (binary data, for example, or parts of a MIME multipart envelope). Example 2-18 references information contained within an external XML document.

Example 2-18. A reference to an external document

```
<person name='joe smith'>
  <address href='http://acme.com/data.xml#joe_smith' />
</person>
```

2.6.2 XML Schemas and xsi:type

The SOAP encoding rule in Section 5.1 states how to express data types within the SOAP envelope, and has caused quite a bit of confusion and challenges for SOAP implementers. Read for yourself:

Although it is possible to use the `xsi:type` attribute such that a graph of values is self-describing both in its structure and that types of its values, the serialization rules permit that the types of values MAY be determinable only by reference to a schema. Such schemas MAY be in the notation described by `XML Schema Part 1: Structures' and `XML Schemas Part 2: Data types' or MAY be in any other notation.

English translation: SOAP defines three different ways to express the data type of an accessor.

1. Use the `xsi:type` attribute on each accessor, explicitly referencing the data type according to the XML Schema specification, as in this example:

```
2.  <person>
3.    <name xsi:type="xsd:string">John Doe</name>
    </person>
```

4. Reference an XML Schema document that defines the exact data type of a particular element within its definition, as in this example:

```
5.  <person xmlns="personschema.xsd">
6.     <name>John Doe</name>
7.  </person>
8.  <!-- where "personschema.xsd" defines the name
       element as type=xsd:string -->
```

9. Reference some other type of schema document that defines the data type of a particular element within its definition, as in this example:

```
10. <person xmlns="urn:some_namespace">
11.   <name>John Doe</name>
12. </person>
13. <!-- where "urn:some_namespace" indicates some
14.       namespace in which the value of name
       elements are strings -->
```

Early SOAP implementations varied in their interpretations of this part of the SOAP specification, causing some rather nasty and annoying integration problems (ironic because SOAP's main goal is to enable interoperability). In particular, the IBM (later Apache) SOAP implementation chose the route of requiring `xsi:type` based typing (forgoing the other two options completely) while the Microsoft SOAP implementation chose to completely ignore the `xsi:type` option in favor of using schemas based on an external service description document. Since neither tool was implemented as a complete implementation of the SOAP

Encoding rules, neither tool was capable of interpreting the data types encoded by the other, even though both were implemented as legal SOAP Encoding schemes. This has, fortunately, since been resolved.

In fact, there has been a large ongoing effort to improve the interoperability between SOAP implementations. For more information about this effort, see the "SOAPBuilders" group at http://groups.yahoo.com/.

2.7 SOAP Data Types

The data types supported by the SOAP encoding style are the data types defined by the "XML Schema data types" specification. All data types used within a SOAP-encoded block of XML must either be taken directly from the XML Schema specification or derived from types therein.

SOAP encoding provides two alternate syntaxes for expressing instances of these data types within the SOAP envelope. Example 2-19 shows two equivalent expressions of an integer equaling the value "36".

Example 2-19. Alternate SOAP encoding syntaxes for typing values

```
<SOAP-ENC:int>36</SOAP-ENC:int>
<value xsi:type="xsd:int">36</value>
```

The first method is what is known as an *anonymous accessor* , and is commonly found in SOAP encoded arrays (as we will see a little later in this chapter). It's "anonymous" because the accessor's name is its type, rather than a meaningful identification for the value. The second approach is the named accessor syntax that we've already seen. Either is valid since they both can be directly linked back to the XML Schema data types.

2.7.1 Multiple References in XML-Encoded Data

The values a program works with are stored in memory. Variables are how programming languages let you manipulate those values in memory. Two different variables might have the same value; for instance, two integer variables could both be set to the value 42. The SOAP XML encoding for this would use single-reference XML, as in Example 2-20.

Example 2-20. Two integer variables set to 42

```
<SOAP-ENC:int>42</SOAP-ENC:int>
<SOAP-ENC:int>42</SOAP-ENC:int>
```

Sometimes, though, you need to indicate that two separate variables are stored in the same piece of memory. For instance, if this subroutine call is going to be XML encoded for SOAP, you'll need to identify the first and second parameters as being the same:

```
tweak(&i, &i);
```

You do this with Section 5's encoding rules using multiple-reference types. That is, you use the `id` attribute to name the value in `i`, then use the `href` attribute to identify other occurrences of that value, as in Example 2-21.

Example 2-21. Multiple-reference to indicate two parameters are the same

```
<value xsi:type="xsd:int" id="v1">42</value>
<value href="#v1" />
```

It's important to understand that even though "SOAP" originally stood for "Simple Object Access Protocol," it actually has no concept of what an object is. To SOAP, everything is data encoded into XML. Therefore there is no such thing as an "object reference" in SOAP. Rather, SOAP Section 5 Encoding specifies a set of rules for transforming an object into XML representing that object. All references to that object that must also be encoded would be done through the use of the `id` and `href` attributes.

Given Example 2-22, the SOAP encoded serialization of the `Person` object might look something like Example 2-23.

Example 2-22. Java code to construct an object

```
Address address = new Address(  );
Person person   = new Person(  );
person.setAddress(address);
```

Example 2-23. SOAP serialization of the object

```
<Person>
   <Address href="#address1" />
</Person>
<Address id="address1" />
```

2.7.2 Structs, Arrays, and Other Compound Types

It was mentioned previously that the difference between an array and a struct in SOAP is that in an array, each accessor in the group is differentiated only by its ordinal position in the group, whereas in the struct, each accessor is differentiated by name. This was shown in Example 2-15.

Even though many programming languages regard strings as an array of bytes, SOAP does not. A string is represented with the string data type, rather than as an array of bytes. If you do have a collection of bytes that you want to ship around, and those bytes do not represent a text string, SOAP Section 5 Encoding decrees that you should use a base64 string, as defined by the XML Schemas specification. The proper serialization of an array of arbitrary bytes, then, is shown in Example 2-24.

Example 2-24. A SOAP-encoded array of bytes

```
<some_binary_data xsi:type="SOAP-ENC:base64">
    aDF4JIK34KJjk3443kjlkj43SDF43==
</some_binary_data>
```

Regular arrays, however, are indicated as accessors of the type `SOAP-ENC:Array`, or a type derived from that. The type of elements that an array can contain is indicated through the use of the SOAP defined `arrayType` attribute, shown in Example 2-25.

Example 2-25. The arrayType attribute

```
<some_array xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="se:string[3]">
   <se:string>Joe</se:string>
   <se:string>John</se:string>
   <se:string>Marsha</se:string>
</some_array>
```

Note the `[3]` appended to the end of the data type value on the `arrayType` attribute. The square brackets (`[]`) indicate the dimensions of the array, while the numbers internally represent the number of elements per dimension. In other words, `[3]` indicates a single dimension of 3 elements, while `[3,2]` indicates a two dimensional array of three elements each. SOAP Encoding supports an unlimited number of dimensions per array in addition to allowing arrays of arrays. For instance, an `arrayType` of `xsd:string[2][]` indicates an unbounded array of single dimensional string arrays, each of which contains two elements.

In Example 2-26, the `data` accessor is an array that contains both of the `names` arrays.

Example 2-26. A two-dimensional array

```
<data xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:string[2][]">
   <names href="#names-1"/>
   <names href="#names-2"/>
</data>
<names id="names-1" xsi:type="SOAP-ENC:Array"
       SOAP-ENC:arrayType="xsd:string[2]">
    <name>joe</name>
    <name>john</name>
</names>
<names id="names-2" xsi:type="SOAP-ENC:Array"
       SOAP-ENC:arrayType="xsd:string[2]">
    <name>mike</name>
    <name>bill</name>
</names>
```

Multidimensional arrays, expressed as XML, are syntactically no different than a regular single-dimension array, with the exception of the value indicated by the `arrayType` attribute. For example, a two-dimensional array of two strings is nearly identical to a one-dimensional array of four strings (shown in Example 2-27).

Example 2-27. Comparison of two-dimensional and one-dimensional arrays

```
<names xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:string[2,2]">
    <name xsi:type="xsd:string">a1d1</name>
    <name xsi:type="xsd:string">a2d1</name>
    <name xsi:type="xsd:string">a1d2</name>
    <name xsi:type="xsd:string">a2d2</name>
</names>

<names xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:string[4]">
    <name xsi:type="xsd:string">a1d1</name>
    <name xsi:type="xsd:string">a2d1</name>
    <name xsi:type="xsd:string">a3d1</name>
    <name xsi:type="xsd:string">a4d1</name>
</names>
```

The value of the `arrayType` attribute distinguishes the true nature of the serialized array.

2.7.3 Partially Transmitted Arrays and Sparse Arrays

SOAP Encoding also includes support for *partially transmitted arrays* and *sparse arrays* through a set of additional attribute definitions.

A partially transmitted array is one in which only part of the array is serialized into the SOAP envelope. This is indicated through the use of the `SOAP-ENC:offset` attribute that provides the number or ordinals counting from zero to the first ordinal position transmitted. In other words, if you have a single-dimensional array of five elements, and you want to transmit only the last two, you would use the syntax in Example 2-28.

Example 2-28. Using SOAP-ENC:offset for partially transmitted arrays

```
<names xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:string[5]"
       SOAP-ENC:offset="[2]">
    <name>Item 4</name>
    <name>Item 5</name>
</names>
```

Sparse arrays represent a grid of values with specified dimensions that may or may not contain any data. For example, if you have a two-dimensional array of ten items each, but only the elements at position `[2,5]` and `[5,2]` contain data, the serialization in Example 2-29 would be appropriate.

Example 2-29. SOAP serialization of sparse arrays

```
<names xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:string[10,10]">
    <name SOAP-ENC:position="[2,5]">data</name>
    <name SOAP-ENC:position="[5,2]">data</name>
</names>
```

2.7.4 Null Accessors

In the sparse array example, the absence of an accessor indicates that the value of the accessor is either null or some other default value. One problem with this is the fact that the receiver of the message has no real way of knowing whether the value of the accessor really was null, or if the sender just failed to serialize the message properly.

If the receiver expects to find the accessor in the message, a better method of indicating whether an accessor contains a null value would be to use the XML Schema defined `xsi:nil="true"` attribute:

```
<name xsi:type="xsd:string" xsi:nil="true" />
```

This allows you to be far more expressive in your encoding of application data and eliminates confusion over the significance of missing elements.

2.8 SOAP Transports

As mentioned before, SOAP fits in on the web services technology stack as a standardized packaging protocol layered on top of the network and transport layers. As a packaging protocol, SOAP does not care what transport protocols are used to exchange the messages. This makes SOAP extremely flexible in how and where it is used.

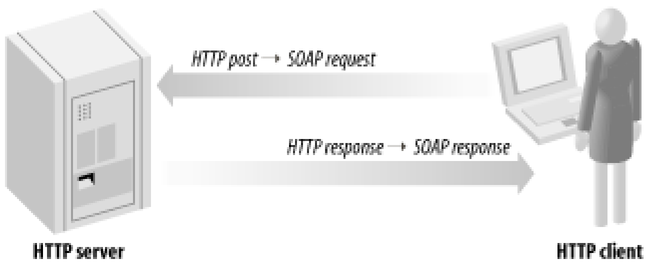
As an illustration of this flexibility, SOAP::Lite—the Perl-based SOAP web services implementation written by Pavel Kulchenko—supports the ability to exchange SOAP messages through HTTP, FTP, raw TCP, SMTP, POP3, MQSeries, and Jabber. We'll show SOAP over Jabber in Chapter 3.

2.8.1 SOAP over HTTP

Because of its pervasiveness on the Internet, HTTP is by far the most common transport used to exchange SOAP messages. The SOAP specification even goes so far as to give special treatment to HTTP within the specification itself—outlining in specific detail how the semantics of the SOAP message exchange model map onto HTTP.

SOAP-over-HTTP is a natural match with SOAP's RPC (request-response) conventions because HTTP is a request-response-based protocol.The SOAP request message is posted to the HTTP server with the HTTP request, and the server returns the SOAP response message in the HTTP response (see Figure 2-5).

Figure 2-5. SOAP request messages are posted to the HTTP server and response messages are returned over the same HTTP connection



Example 2-30 and Example 2-31 illustrate an HTTP request and HTTP response messages that contain a SOAP message.

Example 2-30. HTTP request containing a SOAP message

```
POST /StockQuote HTTP/1.1
Content-Type: text/xml
Content-Length: nnnn
SOAPAction: "urn:StockQuote#GetQuote"

<s:Envelope xmlns:s="http://www.w3.org/2001/06/soap-envelope">
  ...
</s:Envelope>
```

Example 2-31. HTTP response containing a SOAP message

```
HTTP/1.1 200 OK
Content-Type: text/xml
Content-Length: nnnn

<s:Envelope xmlns:s="http://www.w3.org/2001/06/soap-envelope">
  ...
</s:Envelope>
```

The `SOAPAction` HTTP header is defined by the SOAP specification, and indicates the intent of the SOAP HTTP request. Its value is completely arbitrary, but it's intended to tell the HTTP server what the SOAP message wants to do before the HTTP server decodes the XML.

Servers can then use the `SOAPAction` header to filter unacceptable requests.

2.8.2 Contentious Issues

The conventions for sending SOAP over HTTP have always caused difficulty in the SOAP development community. There are a number of issues that seem to come up time and again. Among these are:

Should SOAP really use HTTP port 80 or should a SOAP-specific port be used?

> Because SOAP messages masquerade as traditional web traffic on port 80, firewalls generally pass them straight through. Obviously, security administrators may have a problem with this. There are no requirements that SOAP over HTTP must use port 80, but many people use it specifically to avoid being filtered by firewalls.

Is the SOAPAction header really useful?

> Because its value is arbitrary, there's no way for a server to always know the intent of a request without parsing the XML. This is an issue that has been debated ever since the `SOAPAction` header was first introduced in SOAP Version 1.1. The W3C working group that is standardizing SOAP is leaning towards deprecating the `SOAPAction` header in the next version of the protocol.

When a client fault occurs while processing a SOAP message, should the server send a HTTP 500 "Server Error" back to the client or a HTTP 200 "OK" response with a SOAP fault included?

> This is an interesting question of semantics. A client fault in SOAP is obviously an application level error, and not the result of a server error. The HTTP 500 Server Error response however, is the default response required for all SOAP faults, regardless of the fault code. The general consensus on this question has been that consistency is most important. Despite the fact that client fault types are not Server Errors, the 500 Server Error code is still the right response when HTTP is used for the transport.

Should a SOAP specific URL scheme be used rather than the traditional http:// scheme used for web pages?

> This question, like the one dealing with the use of port 80, directly addresses the question of whether or not SOAP web services should masquerade as more traditional HTTP-based services. Some have maintained that a new `soap://` URL scheme is required. Microsoft's SOAP Routing Protocol even goes so far as to define such a scheme.

While HTTP is the most popular transport for SOAP message, it is not without problems. HTTP was not designed as a transport for XML messages, and there are times when the two protocols don't mesh perfectly. That said, it remains the most popular transport for SOAP, although Microsoft's .NET makes heavy use of SOAP-over-Instant Messaging and this may challenge HTTP's supremacy.